Équipe A* (208)

PolyDiff
Protocole de communication

Version 1.8

Historique des révisions

Date	Version	Description	Auteur
2024-01-14	1.0	Ajouter la description des paquets envoyés par le client et le serveur pour les fonctionnalités déjà existantes	Zarine Ardekani
2024-01-15	1.1	Rédaction de l'introduction	Fatima Zohra Oulaidi
2024-01-15	1.2	Rédiger la description des paquets associés aux canaux de discussion, au compte utilisateur, aux modes de jeu, à la vue d'administration et à la reprise vidéo	Zarine Ardekani
2024-01-20	1.3	Rédiger la description des paquets associés aux observateurs, à la personnalisation de l'application et au système d'amis	Zarine Ardekani
2024-01-22	1.4	Rédiger la description des paquets associés aux nouvelles fonctionnalités inventées.	Zarine Ardekani
2024-01-27	1.5	Réviser et compléter le protocole de communication	Zarine Ardekani
2024-01-28	1.6	Révision du contenu du document et de la qualité du français	Jeremy Rouillard
2024-01-30	1.7	Révision et ajustement de la description des paquets pour refléter le travail fait à date sur le serveur.	Zarine Ardekani Lucas Bouchard
2024-01-30	1.8	Réviser encore et ajuster encore pour être cohérent avec le code.	Zarine Ardekani Lucas Bouchard

Table des matières

Protocole de communication

1. Introduction

Ce document décrit le mode de communication utilisé pour les interactions entre les applications clientes (une application mobile Android et une application de bureau) et le serveur, dans le cadre de la poursuite du projet « PolyDiff: Trouver les différences », qui a été développé au cours d'un semestre précédent. Le présent document est organisé en deux sections : *Communication client-serveur* et *Description des paquets*. La première section décrit le moyen de communication sélectionné entre les clients et le serveur au sein de notre projet tout en fournissant une justification pour nos choix. La deuxième section détaille quant à elle les divers types de paquets qui sont utilisés lors du protocole de communication. Ces détails sont présentés de manière structurée dans un tableau, comprenant le nom de l'événement, le contenu du paquet, la réponse du serveur, la description du paquet, ainsi que d'autres informations pertinentes s'il y a lieu. En bref, le contenu de ce document inclut des prédictions faites en se basant sur la structure du code de l'application web réalisée précédemment puisque les clients et le serveur sont en cours de développement.

2. Communication client-serveur

Comme vous le verrez par la suite, nous avons choisi deux moyens de communications distincts : les WebSockets et les requêtes HTTP. Pour résumer leurs avantages propres, les WebSockets permettent une communication bidirectionnelle entre le client et le serveur et rapide. Quant à elles, les requêtes HTTP offrent une solution unidirectionnelle et plus légère où uniquement le client peut engendrer une requête auquelle le serveur peut répondre. Dans notre cas, les WebSockets sont très utiles dans les situations où le client doit être mis à jour en temps réel par exemple dans une partie de jeu ou dans un salon de messagerie . Concernant les requêtes HTTP, elles sont utiles quand le client doit effectuer une action spécifique et engendrée par l'utilisateur comme par exemple bloquer un utilisateur ou obtenir les données pour une reprise vidéo. Nos choix technologiques pour les différents groupes de fonctionnalités sont détaillés dans le tableau à la page suivante (Tableau 1).

Lorsque nous devons envoyer une image dans un paquet, l'image est encodée en une chaîne de caractères en base 64. Cet encodage permet un affichage facile dans une page HTML ainsi que la communication des données de l'image entre le client et le serveur.

Pour plus de lisibilité et de cohérence, nos requêtes HTTP respectent le style d'architecture REST (*REpresentational State Transfer*). Lorsqu'une identification est nécessaire, un jeton d'identification obtenu lors de la connexion est inclus dans les requêtes au serveur.

Fonctionnalités	Explication
Administration des jeux	Les websockets, car la liste des cartes de jeu pourrait être mise à jour à tout moment.
Sélection de partie	Les websockets, car la liste des jeux et des joueurs en attente doit être mise à jour.
Déroulement d'une partie	Les websockets, car le serveur doit pouvoir envoyer divers événements pendant la partie.
Clavardage	Les requêtes HTTP pour ajouter ou supprimer un canal de discussion. Les websockets pour l'envoi de message et la mise à jour de la liste des canaux, le serveur doit pouvoir informer les clients des nouveaux messages et des changements dans la liste des canaux de discussion.
Création de jeu	Les requêtes HTTP, car le client initie la création de jeu et qu'il ne peut envoyer qu'une requête unidirectionnelle simple à cet effet.
Compte utilisateur	Les requêtes HTTP pour la connexion, la création, l'obtention des détails du compte et la modification du compte puisque ce sont des actions qui peuvent utiliser des communications unidirectionnelles. On suppose que l'utilisateur doit rafraîchir la page pour avoir les mises à jour du profil d'un autre utilisateur.
Reprise vidéo	Les requêtes HTTP pour la création, la suppression et l'obtention du contenu d'une reprise vidéo, car c'est une action initiée par l'utilisateur. Les websockets pour obtenir la liste des reprises vidéo sauvegardées, car cette liste doit pouvoir être mise à jour à tout moment.
Observation de partie	Les websockets, car le serveur doit pouvoir envoyer des événements liés aux actions des observateurs.
Personnalisation de l'application	Les requêtes HTTP, car les changements de thème et de langue sont uniquement initiés par le client.
Système d'amis	Les requêtes HTTP pour demander à bloquer ou débloquer un utilisateur. Les websockets pour les fonctionnalités liées à la liste d'amis, car celle-ci doit pouvoir être mise à jour à tout moment pour refléter des changements dans le profil des autres utilisateurs ou pour refléter un blocage/déblocage.

Tableau 1. Communication HTTP et websockets

3. Description des paquets

3.1 Tables des paquets WebSocket envoyés par le client

Nom de l'événement	Contenu	Description et autres informations	Réponse(s) du serveur
Administration des jeux			
all_game_cards		Obtenir toutes les cartes de jeu.	all_game_cards

delete_card	{
	cardId: string
	}
		Supprimer une carte de jeu.	card_delete_response

delete_all_cards		Demander à supprimer toutes les cartes de jeu.	
Sélection de partie			
joinable_game_cards		Obtenir les IDs des jeux avec partie en attente.	joinable_game_cards

create_game	{
	gameMode: GameMode;
	cardId? : string;
	accessibility: AccessType;
	timerTime: number;
	penaltyTime: number;
	gainedTime: number;
	canCheat: boolean;
	}
		Envoyer des informations au serveur lorsqu'un joueur veut créer une partie en mode classique ou en mode temps limité.
		Pour le mode temps limité, cardID est undefined.
		AccessType est un enum qui peut prendre les valeurs suivantes : {EVERYONE, FRIENDS, FRIENDS_OF_FRIENDS}
			create_game_response

request_to_play	{
	gameId? : string;
	}
		Envoyer des informations au serveur lorsqu'un joueur veut rejoindre une partie en mode classique ou en mode temps limité.	response_to_play_request

choose_team	{
	teamId: string
	}
		Rejoindre une équipe pour une partie classique en équipe avec l'ID de l'équipe.	

Déroulement d'une partie			
is_playing		Demander au serveur si une partie est en cours.	is_playing

send_click	{
	gameId : string;
	x: number;
	y: number;
	}
		Envoyer des coordonnées du clic et de l'ID du jeu en cours. Cet événement avertit le serveur lorsqu'un joueur clique sur une des deux images (soit l'image originale ou l'image modifiée).	click_personal

get_cheats	{
	gameId: string;
	}
		Obtenir les images de triche.	cheat

leave_game	{
	gameId: string;
	}
		Cet événement survient lorsqu'un joueur abandonne une partie en cours ou si un joueur quitte l'attente d'autres joueurs pour une partie en mode classique ou pour une partie en mode temps limité.	

extra_time	{
	gameId: string;
	}
		Obtenir du temps supplémentaire en mode temps limité lorsque l'utilisateur secoue la tablette et crie.	

rate_game	{
	gameId : string;
	like: boolean;
	}
		Évaluer la partie qui vient de se terminer: l'utilisateur peut aimer (like) ou ne pas aimer (dislike) la partie.	

Clavardage			

send_chat_message	{
	channelId : string
	message : string
	}
		Envoyer le message écrit par le joueur et l'ID du canal.	

get_all_channels		Obtenir les noms et les IDs de tous les canaux de discussion existants.	channel_update
Reprise vidéo			
get_video_list		Obtenir la liste des reprises vidéo enregistrées dans le compte de l'utilisateur.	video_list
Observation de partie			

watch_game	{
	gameId: string;
	}
		Rejoindre une partie en tant qu'observateur.	

interact	{	Interagir avec un ou tous les	

	gameId: string;
	zone: {
	start: Coordinates;
	end: Coordinates;
	};
	forPlayerName?:
	string;
	}
		joueurs d'une partie en mode observateur.
		Quand aucun nom de joueur n'est spécifié, on interagit avec tous les joueurs.
		Coordinates contient un nombre x et un nombre y.	

play_sound	{
	gameId : string;
	soundId: string;
	}
		Jouer un son qui sera entendu par les joueurs.	

Système d'amis			

start_friend_request	{
	username: string;
	}
		Faire une demande d'ami à l'utilisateur ayant le pseudonyme fourni.	

search_users	{
	searchCriteria: string;
	}
		Rechercher des utilisateurs avec un argument de recherche.	search_users_result

3.2 Tables des paquets WebSocket envoyés par le serveur

Nom de l'événement	Contenu	Description et autres informations	En réponse à
Administration des jeux			

all_game_cards	{
	id: string;
	name: string;
	difficulty: number;
	classicSoloBestTimes: BestTimes;
	classic1v1BestTimes: BestTimes;
	originalImage: string;
	rating: number;
	}[ ]
		Envoyer toutes les cartes de jeu. BestTimes contient trois objets BestTime qui contiennent chacun les informations suivantes:
		{
		name: string;
		time: TimeConcept;
		}
			all_game_cards

game_card	{
	id: string;
	name: string;
	difficulty: number;
	classicSoloBestTimes: BestTimes;
	classic1v1BestTimes: BestTimes;
	originalImage: string;
	}
		Envoyer aux clients une carte de jeu qui vient d'être créée. BestTimes contient trois objets BestTime qui contiennent chacun les informations suivantes:
		{
		name: string;
		time: TimeConcept;
		}	

card_delete_response	{
	deleted: boolean;
	}
		Booléen indiquant si la carte a été supprimée.	delete_card

Sélection de partie			

joinable_game_cards	{
	gameIds: string[];
	}
		Liste des IDs de jeux avec partie en attente.	joinable_game_cards

create_game_response	{ responseType: GameConnectionAttemptResponseType; gameName: string; playerNbr: number; startingIn: number; originalImage: string; modifiedImage: string; time: number; gameId: string; difficulty: Difficulty; differenceNbr: number; hostName: string;	Cet événement contient les informations nécessaires pour commencer une partie en mode classique ou en mode temps limité. GameConnectionAttemptResponseType est un enum dont les valeurs possibles sont starting, pending, cancelled et rejected.	create_game

	gameValues: GameValues; }		

response_to_play_request	{
	responseType:
	GameConnectionAttemptResponseType;
	gameName: string;
	playerNbr: number;
	startingIn: number;
	originalImage: string;
	modifiedImage: string;
	time: number;
	gameId: string;
	difficulty: Difficulty;
	differenceNbr: number;
	hostName: string;
	gameValues:
	GameValues;
	}
		Cet événement contient les informations nécessaires pour commencer une partie en mode classique ou en mode temps limité.
		GameConnectionAttemptResponseType est un enum dont les valeurs possibles sont starting, pending, cancelled et rejected.
			request_to_play

available_teams	{
	teamId: string;
	players: {
	name: string;
	avatar: string;
	}[]
	playerNames: string[];
	}[]
		Envoyer les noms et les avatars des joueurs dans les équipes disponibles pour une partie classique en équipe.	

Déroulement d'une partie			

is_playing	{
	isPlaying: boolean;
	}
		Envoyer un booléen indiquant si la partie a bien commencé ou non. Si une erreur est survenue, le booléen sera à false et le joueur sera transporté à la page d'accueil. Sinon, le jeu continue normalement.	is_playing

player_status	{ playerConnectionStatus: PlayerConnectionStatus; user?: SimpleUser; playerNbr?: number; }[]	Contient les informations des autres joueurs. Si playerConnectionStatus est Left et le mode de jeu est temps limité à deux joueurs, la vue de jeu se transforme en partie solo en mode temps limité. PlayerConnectionStatus est un enum dont les valeurs possibles sont AttemptingToJoin, Left et Joined. SimpleUser contient les informations suivantes: {name: string, id: string}	

time	{
	time: number;
	}
		Contient le temps en secondes du chronomètre.
		Cet événement sert à envoyer le temps à afficher pendant la partie. Cet événement est envoyé à chaque 0.25 seconde et tient compte de la pénalité d'utilisation d'indices, ainsi que du bonus de temps obtenu à la suite d'une différence trouvée.	

click_personal	{
	valid: boolean;
	playerName?: string;
	penaltyTime?: number;
	
	differenceNaturalOverlay?: string;
	differenceFlashOverlay?: string;
	}
		Cet événement indique la validité du clique du joueur et donne accès aux flashOverlay et naturalOverlay si le clic est valide.	send_click

click_enemy	{
	valid: boolean;
	playerName?: string;
	penaltyTime?: number;
	
	differenceNaturalOverlay?: string;
	differenceFlashOverlay?: string;
	}
		Cet événement survient lorsqu'un autre joueur effectue un clic. Cet événement indique la validité du clic et donne accès aux flashOverlay et naturalOverlay si le clic est valide.	

cheat	{
	cheatImages: string[];
	}
		Liste des images de flash en base 64.	get_cheats

cheat_index	{
	cheatIndex: number;
	}
		Index de l'image de triche à retirer parce que la différence a été trouvée.	

next_card	{
	name: string;
	originalImage: string;
	modifiedImage: string;
	nbDifferences: number;
	}
		Cet événement permet d'obtenir les prochaines images (originales et modifiées) pour le mode temps limité.	

deserter	{
	username: string;
	}
		Cet événement est généré lorsqu'une personne quitte un partie. L'événement avertit les autres joueurs qu'un joueur a quitté la partie.	

endgame	{
	finalTime?: number;
	newBestTimes?: BestTimes;
	players: PlayerRecord[];
	}
		Contient les informations de fin de partie, soit les informations des joueurs, le temps final mis pour compléter la partie s'il y a lieu et les nouveaux meilleurs temps de jeu s'il y a lieu.
		PlayerRecord contient les informations suivantes :
		{
		name: string;
		winner: boolean;
		deserter: boolean;
		}
		BestTimes contient trois objets BestTime qui contiennent chacun les informations suivantes:
		{
		name: string;
		time: TimeConcept;
		}	

Clavardage			

chat_message	{
	sender: {
	name: string;
	avatar: string;
	};
	message: string;
	}
		Cet événement indique au client l'auteur du message et lui fournit le message à afficher.	

global_message	{
	message: string;
	}
		Envoyer un message global.	

channel_update	{
	type: ChannelUpdateType;
	channelId: string;
	channelData?: ChannelData;
	}[]
		Envoyer aux clients la liste des canaux de discussion et les mises à jour lorsqu'un canal est créé ou supprimé.
		
		ChannelUpdateType est un enum qui peut prendre l'une ces trois valeurs:
		{INFO, CREATE, DELETE}
		
		ChannelData contient:
		{
		name : string;
		owner: string;
		members: string[];
		}
			get_all_channels

Reprise vidéo			

video_list	{ replayId: string; cardName: string; public: boolean; }[]	Envoyer la liste des reprises vidéos enregistrées par un utilisateur. Cet événement est renvoyé lorsque la liste des reprises vidéo de l'utilisateur change.	get_video_list
Observation de partie			

watched_games	{
	gameIds: string[];
	}
		Liste des IDs de jeux avec au moins un observateur.	

watchers	{
	nbWatchers: number;
	}
		Indiquer combien d'observateurs observent la partie.	

interaction	{
	zone: {
	start: Coordinates;
	end: Coordinates;
	};
	color: string;
	}
		Envoyer les coordonnées d'une interaction d'un observateur et la couleur dans laquelle elle doit être affichée.	

play_sound	{
	soundId: string;
	}
		Envoyer aux joueurs l'ID du son joué par un observateur.	

Système d'amis			

start_friend_request	{
	username: string;
	}
		Envoyer une demande d'ami en précisant le pseudonyme de l'utilisateur faisant la demande.	

search_users_result	{
	users: string[];
	}
		Envoyer le résultat de la recherche de joueurs dans l'écosystème du jeu.
		Contient une liste de pseudonymes.
		Cet événement est envoyé lorsqu'une recherche est effectuée ou lorsqu'il y a un changement à la liste d'utilisateurs après le début de la recherche.
			search_users

friendship_update	{ type: FriendshipUpdateType; username: string; friendData?: FriendData; }[]	Envoyer les mises à jour de la liste d'amis. FriendshipUpdateType est un enum qui peut prendre l'une ces trois valeurs: {INFO, JOINED, LEFT}	

3.3 Table des paquets HTTP

Méthode	Route	Corps	Code de retour	Corps de la réponse
Administration des jeux				

POST	/admin/login	{
		password: string;
		}
			200 (succès)
			400 (échec)
				En cas de succès :
				{
				sessionToken: string;
				}
				
				En cas d'échec :
				{
				errorMessage: string;
				}

Clavardage				

POST	/channels	{
		name: string;
		}
			201 (succès)
			400 (échec)	

DELETE	/channels/{id}		200 (succès) 400 (échec) 404 (canal introuvable)	
Création de jeu				

POST	/cards/validate	{
		originalImage: string;
		modifiedImage: string;
		range: number,
		}
			200 (succès)
			400 (échec)
				En cas de succès :
				{
				differenceNbr: number;
				difficulty?: Difficulty;
				differenceImage?: string;
				}
				
				En cas d'échec :
				{
				errorMessage: string;
				}

POST	/cards	{
		originalImage: string;
		modifiedImage: string;
		range: number;
		name: string;
		}
			201 (succès)
			400 (échec)	

Compte utilisateur				

POST	/auth/signup	{
		email: string;
		hashedPassword: string;
			201 (succès)
			400 (échec)
				En cas de succès :
				{
				accessToken: string;
				}

		username: string;
		}
				En cas d'échec :
				{
				code: string;
				message: string;
				name: string;
				}

POST	/auth/login	{
		username: string;
		hashedPassword: string;
		}
			200 (succès)
			400 (échec)
				En cas de succès :
				{
				accessToken: string;
				}
				
				En cas d'échec :
				{
				code: string;
				message: string;
				name: string;
				}

GET	/users/{username}		200 (succès)
			400 (échec)
			404 (compte introuvable)
				En cas de succès :
				{
				username: string;
				avatar: string;
				elo: number;
				biography: string;
				friends: string[];
				}
				
				En cas d'échec :
				{
				errorMessage: string;
				}

GET	/users/me		200 (succès)
			400 (échec)
				En cas de succès :
				{
				username: string;
				email: string;
				avatar: string;
				biography: string;
				generalGameStatistics: GeneralGameStatistics;
				friends: string[];
				blockedUsers: string[];
				interfacePreferences: InterfacePreferences;
				pendingFriendRequests: PendingFriendRequest[];
				}
				
				GeneralGameStatistics contient les informations suivantes : {
				elo: number;
				gamesPlayed: number;

				gamesWon: number;
				averageDifferencesFound: number;
				averageTime: number;
				}
				
				InterfacePreferences contient les informations suivantes: {
				theme: string;
				language: string;
				soundIds: {
				error: string;
				differenceFound: string;
				}
				
				En cas d'échec :
				{
				errorMessage: string;
				}

PUT	/users/{username}	{
		username: string;
		avatar: string;
		biography: string;
		}
			200 (succès)
			400 (échec)
			404 (compte introuvable)
				En cas d'échec :
				{
				validUsername: boolean;
				errorMessage: string;
				}

DELETE	/users/{username}		200 (succès) 400 (échec) 404 (compte introuvable)	

POST	/users/reset-password	{
		email: string;
		}
			200 (succès)
			400 (échec)	

PATCH	/users/password	{
		email: string;
		newPassword: string;
		code: string;
		}
			200 (succès)
			400 (échec)	

Reprise vidéo				

POST	/videos	{
		gamedId: string;
		replayActions: ReplayAction[];
		}
		
		ReplayAction contient les informations suivantes :
			201 (succès)
			400 (échec)	

		{
		startTime: number;
		endTime: number;
		action: () => void;
		}		

GET	/videos/{id}		200 (succès)
			400 (échec)
			404 (reprise introuvable)
				En cas de succès :
				{
				replayActions: ReplayAction[];
				gameName: string;
				playerNbr: number;
				originalImage: string;
				modifiedImage: string;
				time: number;
				difficulty: Difficulty;
				differenceNbr: number;
				hostName: string;
				gameValues: GameValues;
				}
				
				En cas d'échec :
				{
				errorMessage: string;
				}

DELETE	/videos/{id}		200 (succès) 400 (échec) 404 (reprise introuvable)	
Personnalisation de l'application				

PUT	/preferences	{
		theme: string;
		language: string;
		soundIds: {
		error: string;
		differenceFound: string;
		}
		}
			200 (succès)
			400 (échec)	

Système d'amis				
POST	/users/blocked/{username}		200 (succès) 400 (échec) 404 (utilisateur introuvable)	
DELETE	/users/blocked/{username}		200 (succès) 400 (échec) 404	

			(utilisateur introuvable)	

POST	/user/friends/answer/{username}	{
		accept: boolean;
		}
			200 (succès)
			400 (échec)
			404 (utilisateur introuvable)	

DELETE	/user/friends/{username}		200 (succès) 400 (échec) 404 (ami introuvable)	